BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

COMPUTER SCIENCE IN ENGLISH SPECIALIZATION

DIPLOMA THESIS

An Interactive Platform for Domain-Specific Data Analysis and Visualization

Supervisor
Assist prof. dr. Károly Simon

Author
Alpár Cseke

2020

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA

FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ ENGLEZĂ

LUCRARE DE LICENŢĂ

O platformă interactivă pentru analiza si vizualizarea unor date specifice

Conducător ştiinţific
Lector dr. Simon Károly

Absolvent
Cseke Alpár

2020

Abstract

The purpose of the application to be presented is managing personal and historical data related to priests and the parishes they served at. The website offers a user-friendly platform for recording new data or modifying preexisting entries. By providing not only a wide range of filtering tools and presentation options, but also a clear browsing experience, it aims to help in revealing correlations in the dataset, encouraging and helping research in the field.

The platform also features a role-based multi-tier user management system where new users must request access via a registration form evaluated by the administrators of the website. This is in order to protect the personal information stored in the system and fine-tune its accessibility.

The paper first provides a brief introduction to the project by discussing its motivations and requirements. The second chapter presents the theoretical background of the used technologies and the architecture of the application, while the next part demonstrates some implementation details. Chapter 4 discusses the tools and methodologies of the development process, then in the fifth chapter the usage of the application is presented. The thesis concludes with a verdict about the accomplishments of the project and with an overview of further development plans.

This work is the result of my own activity. I have neither given nor received unauthorized assistance on this work.

Alpár Cseke

Contents

Introduction

In this age of information technology, everything revolves around data. The collecting or generating of raw data is the backbone of every interdisciplinary utilization of the tools of computer science. Even so, the accent always remains not on the data itself, but on what is achieved using that particular set of values, or in other words, how it is transformed into information.

A database consisting of records related to a specific field is not much of a value for a community, if the dataset is not available to anyone, or if it is not easily searchable and explorable via a well thought-out browsing interface. Solving this issue is the motivation behind the project presented in this thesis.

The database managed by the software system is a growing, rather valuable dataset consisting of various information about the Protestant priests of Transylvania from 1800 up to the present days.

Many low-code or no-code platforms provide the possibility of generating an application that is connected to a given database, such as Google's App Maker [30] or Microsoft's Power Apps [33]. These offer all the essential operations that can be executed on data records, summed up by the CRUD acronym: creating, reading, updating and deleting them. Unfortunately, all the available CRUD application generators share two significant drawbacks.

The first of these problems is the limited support for fine-tuning the access to different functionalities of the application. Since the sensitive nature of the data necessitates secure access, the application to be presented features five distinct user roles. All of these exhibit different access rights to the data and to the related components. Creating an account with even the lowest of these ranks requires the approval of the website's administrators, who are also in control of the access levels assigned to the accounts.

The second drawback is an inherent one. By their definition, general platforms cannot feature unique functionalities that are based on the specific properties of the underlying data. In

this case, the provided dataset offers unique information about its subject, making it a valuable basis for different types of researches. The software platform aims to provide two different approaches to this, both reliant upon an extensive data filtering module: biographical research by searching for individual profiles and statistical research assisted by cumulative filtering and displaying options as part of a data visualization designer component.

As part of the bigger picture, there are a series of individual problems undertaken, such as sending various e-mail notifications during the registration-evaluation process, the internationalization of the website, flexible configuration and continuous deployment, and ensuring data security via authentication and authorization. Another challenge was the initial UX (Uxer Experience) [15] research conducted in order to design the user interface in such a way that the complex functionalities remain easily usable even by technologically less knowledgeable people.

The thesis is structured as follows. The first chapter introduces the reader to the application's core features grouped by the required minimum user ranks. Chapter 2 contains the theoretical backgrounds of the technologies, languages, frameworks and libraries used in the development of the software, while in its third section it elaborates on the underlying architecture. The next chapter discusses some implementation details regarding all of the main architectural layers: the structure of the data and the means of its processing, the backend server's components and the web client's realization. Chapter 4 focuses on development tools and methodologies, including the project management, the testing processes and the external tools used during the development. Finally, the fifth chapter presents the usage of the application through screenshots and a short tour of some of the most important functionalities. The thesis's last part sums up the accomplishments of the project and lists future improvement possibilities.

Acknowledgment must be given to the Codespring company. The project launched during a summer internship through their student mentorship program. Two intern students were working on the development: István Király and the author of this thesis. The development was coordinated by senior mentors: Károly Simon Ph.D., Szilárd-Gábor Mátis and Kincső Tüzes-Bölöni. The project was also part of the faculty's compulsory *Group project* subject, where three additional students of the faculty - Eliézer Béczi, Szilárd Jakab and Dániel Hunyadi - joined the development for a semester. The Protestant Theological Institute of Cluj-Napoca must also be mentioned, from where the idea of the application and the underlying dataset originated. They also provided continuous personal support during the development.

As stated above, the project as a whole is not solely individual work, thus the thesis only focuses on the contributions of the author. Feature-wise, these include the map and list-based search components, the chart designer, the internationalization of the website, and the normalization and presentation of dates. The authentication and authorization parts are also implemented by the thesis's author, with all of its security aspects discussed in the paper, except for password changing. Other tasks carried out by the author cover the initial setup of the technology stack and frameworks, the continuous integration and the deployment of the project. The components implemented by the author's colleague (the profile pages and their editing, the handling of files and pictures, the news-stream on the main page, the password-changing feature and the data preprocessing) are not presented in details. Nevertheless, all functionalities of the application are briefly described to provide a more complete picture of the accomplishments.

The project was presented at the *23rd Transylvanian Students' Scientific Conference (ETDK)* [32], where it obtained the 2nd place in the *Innovative computing products and applications* section.

Chapter 1

Requirements of the application

This chapter provides a non-technical overview of the requirements of the project through listing the core functionalities and elaborating on the purpose of the project.

1.1 The purpose of the project

As stated before, the application is about managing a dataset of priest and parish entities. Aside from the usual data manipulation operations, such as presenting data fields, creating new entities or editing existing ones, the web page includes several more advanced features, which ultimately makes the product stand out.

Another focus point of the application is its security measures. Since the managed data has a very personal and sensitive nature, it is of highest importance to control access to it. This is implemented by a request-approval based registration system, meaning that the creation of any new account ultimately depends on administrators. They are also in control of the access levels of the accounts via assigning different user roles to them.

The rest of the chapter presents the functionalities of the software platform grouped by their required minimum access levels.

1.2 Functionalities by user roles

For a visual representation of the section's content, please refer to Figure 1.1 containing a use case diagram regarding the user roles and the distribution of access rights across all functionalities.

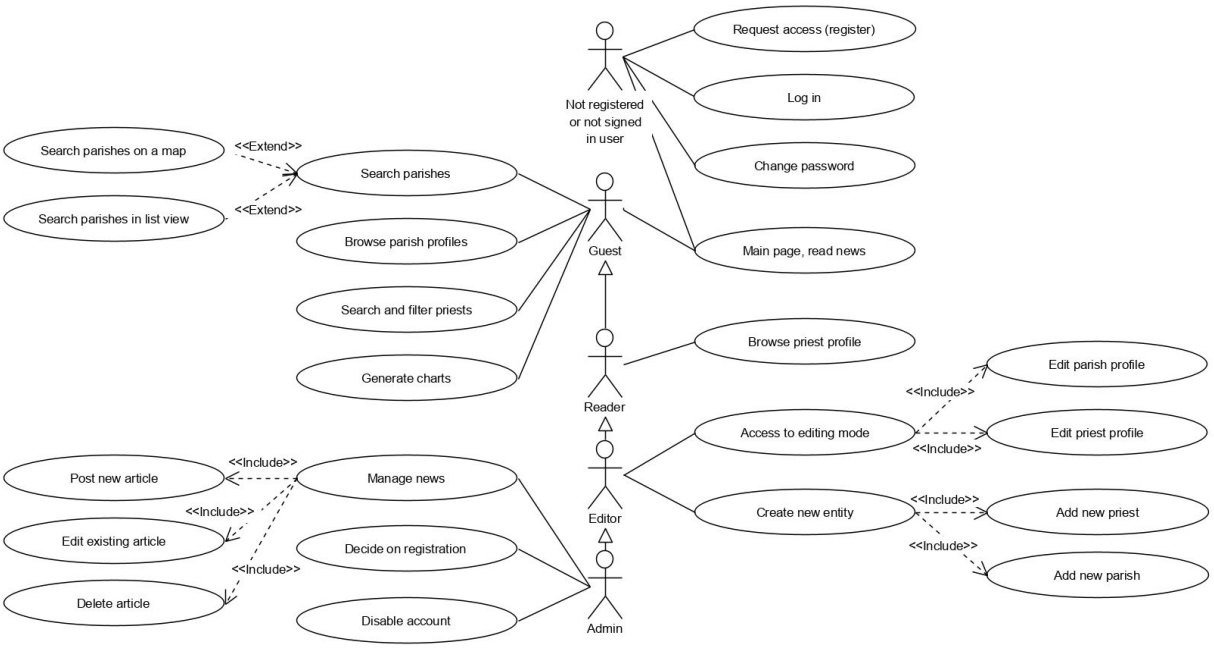

Figure 1.1: Use case diagram containing an overview of the functionalities and presenting the user role hierarchy.

Users who are **not signed in** are only able to access the authentication operations and the main page of the website. This page includes a description about the project and contains a news-stream with the updates and announcements of the website. The aforementioned authentication operations include the login page, the password change requesting and modifying interfaces, and the access request form. The latter only initiates the registration process - by providing a name, an e-mail address, a password and a short self-introduction - the account's creation depends on administrator approval.

Logged in users with the **Guest** role are able to access the searching pages and the chart creator too. When searching priests, only the data shown on the result cards is available, guests cannot navigate to the priest profile pages. However, they can browse the parish profiles with all existing data, which is accessible from the placement list of priests or from the dedicated parish searching components. The latter is available both in list view and on a map interface.

Reader or higher ranked users can benefit from one additional functionality, the priest profile pages, where all available personal data is present. The page is reachable from the parish profiles and naturally from the priest filtering component as well.

The role of **Editors** cover two more feature groups. They can create new data sheets for priests or parishes by providing some basic information about them, if these do not exist yet. In case of matching data, they are redirected to the already existing entity. On the profile

pages, a button will appear for Editors, making it possible to enter editing mode to change fields and manage files and pictures. On the priest profiles, new family members, placements, qualifications, occupations etc. can be added as well.

The **Admin** is the highest user rank. Aside from all previous functionalities, they are granted access to all user data (name, e-mail, role), and they are able to delete existing accounts. However, their most important task is to evaluate registration requests, which also takes place on the admin dashboard. Admins can decide to accept or refuse new accounts, and the user's rank is also determined by them. They can also include the reasoning behind their decision, which will be part of the notification e-mail sent to the user. Last, but not least, only these users can create, edit or delete news on the main page.

There is also an **Owner** rank. Its access rights are the same as the Admins', but there is only one such account and it cannot be deleted, contrary to Admins.

Chapter 2

Technological background

This chapter introduces the reader to all the noteworthy technologies being used in the implementation of the backend server and the web client, as well as the data access layer.

2.1 Technologies of the backend server

The two main components of the software platform's backend part are the MongoDB database providing permanent data storage and the application server itself, implemented in Golang. The section presents not only these, but a handful of other third party Golang libraries as well.

2.1.1 MongoDB

In document-oriented NoSQL databases, such as MongoDB, data is organized in so-called documents instead of the rows and columns known from SQL. In MongoDB, the structure of these documents is similar to the JSON[1] format [35]. It consists of key-value pairs, where the value can be scalar, a list or even an inserted subdocument too. It also supports binary data values for realizing file storage, hence the name BSON (Binary JSON).

The collections of these entities are schema-less, meaning that documents belonging to a collection are not necessarily identical in structure, though they are usually somewhat similar. Between collections, relations are possible to be created by the documents' unique identifiers, however it is not always recommended. Instead, related data should be stored in the same collection and document, as subdocuments. One of MongoDB's significant trait is its ample

[1] JavaScript Object Notation, a lightweight and human-readable data interchange format.

filtering interface, which is capable of carrying out queries based on range filters or regular expressions as well.

2.1.2 Golang

The backend server querying the aforementioned database is implemented in Golang, or simply Go, a performance and simplicity oriented language developed and maintained by Google. Golang is most often compared to the C language, since it does not feature classes and related features such as inheritance, instead it relies only on *structs* as building blocks of the code [11].

It is a compiled language, therefore it offers runtime performance and more rigorous syntax checking. Golang also features garbage collection, while the so-called *goroutines* offer a lightweight and simple solution for multi-threading. Another big differentiator of the language is that it tries to remain simple to use, to understand and to debug. It does not feature classes, any sort of inheritance, generic types and exception handling try-catch schematics.

Despite being a relatively new language, it comes with a wide range of libraries, thanks to Google's kick-start approach [12]. Packages are automatically downloaded and dynamically versioned by a system very similar to JavaScript's node package manager (npm), called Go Modules [6]. Some of the most important extension packages used in the current project are presented in the next subsection.

2.1.3 Auxiliary Golang packages

Gin implements a HTTP[2] based web framework with a router that distributes the incoming calls from the frontend server between the handler functions found in the controller layer [27]. It is one of the most performant HTTP web framework for Golang to date, thanks to its underlying Radix tree[3] data structure. It is also highly configurable with different middleware layers. In the current project these include a custom logger, a session manager, a CORS[4] configuration [20] and a role-based access control interceptor too. The library determines the underlying architecture of the backend server, the router being the base of all communications done with the web client.

[2]Hypertext Transfer Protocol, an application-layer protocol for transmitting hypermedia documents mainly between browsers and web servers.

[3]A space-optimized prefix tree, where only child nodes are merged with their parents.

[4]Cross-Origin Resource Sharing, a mechanism allowing requests from external sources.

Viper is a configuration management library [9]. It is used to retrieve and prioritize between configuration options coming from different inputs in different environments. These external variables can be given via the host operating system's environment variables, docker-compose service parameters or ordinary key-value configuration files. What Viper does is it retrieves all of the configurations from all sources and decides which sources override which ones. This is essential for the backend server, since it is ran in multiple environments during development: locally, in docker containers and in test deployments.

The **Logrus** logging package makes it possible to trace back potential bugs. The logs' color and included background data (timestamp, location) are both customizable, and the configurations can be integrated into other libraries' (e.g.: Gin) internal loggers [7].

The last package to be mentioned in this subsection is the **mongo-go-driver** [34] which acts as a bridge between the backend server and the MongoDB database. It manages a fluid communication between these processes, and translates queries constructed imperatively in Golang into native MongoDB requests.

2.2 Client-side technologies

This section is a compilation of all the basic notions to be understood about the technology stack used to implement the web client part of the software platform.

2.2.1 React

Initially, the **React** library was internally developed and used by Facebook. After its 2013 release, it quickly became the most popular JavaScript-based frontend framework for developing web applications, which is proven by JavaScript's official package manager's, npm's download statistics [16]. To understand its benefits, multiple concepts should be explained.

It is a component-oriented framework, meaning that the elements of the user interface are grouped into so-called components. Not only does this help in structuring the code, but these components can also be reused multiple times across different parts of the application and even in other web pages.

The optional, albeit recommended JSX[5] syntax is also a unique trait. Instead of imperatively

[5]JavaScript XML, where XML stands for Extensible Markup Language

```
class Hello extends React.Component {
  render() {
    return React.createElement(
      'div', null,
      `Hello ${this.props.toWhat}`
    ); }
}
ReactDOM.render(
  React.createElement(
      Hello,
      {toWhat: 'World'},
      null
  ),
  document.getElementById('root')
);
```

(a) Plain JavaScript syntax.

```
class Hello extends React.Component {
  render() {
    return(
      <div>Hello {this.props.toWhat}</div>
    );
  }
}

ReactDOM.render(
  <Hello toWhat="World" />,
  document.getElementById('root')
);
```

(b) JSX syntax.

Listing 2.1: React's two available syntax flairs.

declaring components in plain JavaScript, elements can be structured in a HTML[6]-like manner. In the background, this is realized by calling the constructors of React classes via the *"<></>"* shorthand notation. This makes it possible to use the syntax seen on the right side of Listing 2.1[7] to declare and instantiate components, instead of the one on the left side. This is not only shorter, but is much closer to the HTML design many web developers are used to. When compiled, code written with JSX also takes the form seen on Listing 2.1a, thus there are no performance drawbacks.

The use of React revolves around one-way data flow. Data from upper components is passed to lower tiers via immutable variables, i.e. it "flows down". However, these components cannot directly modify the received data, that is only possible by using special functions also passed down to the lower layers. These functions trigger callbacks into higher layers, therefore actions are said to "flow up", towards the component owning the data.

The Virtual DOM (Document Object Model) is also React-specific. Aside from the browser's static DOM, the JavaScript engine also stores a tree-structured hierarchy of the page's elements. Actions affecting the former triggers the Diffing algorithm in the VDOM, which refreshes only the subcomponents directly influenced by the action [40], instead of whole component subtrees or even the entire web page.

Another unique concept is the *react state*. This is based on that each component's required data is part of it, but in such a way that the resource is located in the lowest possible node of the

[6]Hypertext Markup Language, the standard markup language of web pages.
[7]Source: https://reactjs.org/docs/react-without-jsx.html, visited on: 2020-06-05

virtual DOM, from where all other subcomponents can still reach it with the one-way data flow, in the so-called common root. All variables tied to this react state are being monitored, and in case they change, the affected parts of the components re-render.

2.2.2 Other JavaScript libraries

In essence, **TypeScript** is JavaScript's extension to a typed language. It is a superset of JavaScript, thus any code written in JavaScript is also valid TypeScript code, meaning all of its introduced syntactical elements are optional. Among these are type-annotations and the compile-time type checking. Generic types and interfaces are also available, the latter is mainly used to constrain the input and output data of functions.

Semantic UI is a CSS[8] framework for theming web applications. The package's React-compatible version relies on components written in React instead of static use of CSS class names [44]. This greatly reduces the required development time, since the input variables controlling the different visualization options of these elements are now type-checked and statically analyzed thanks to TypeScript.

The **Axios** library is responsible for sending HTTP requests to the backend server. It uses a promise-based asynchronous system [23], meaning that after a request is sent, the program does not just halt and wait for the answer, but carries on and the potentially incoming answer or error will intercept the application to pass its data to it. Another advantage of the package is that it also integrates with TypeScript, therefore the request-sending functions are generic by implementation and the returned JSON structures are automatically decoded and converted to the given interface types.

Creating the map of the parishes was made possible by **Leaflet** [13]. It connects to the freely available OpenStreetMap[9] service, from where it retrieves the tiles required to assemble the currently displayed map. It is further enhanced by the React-Leaflet-MarkerCluster plugin, which groups markers close to each other into clusters for better visibility, and it also makes possible spreading markers placed on the exact same coordinates by clicking on them.

Reaviz [42] is used for rendering the user-defined diagrams of the chart designer component. The library offers a sizable selection of different chart types which are all deeply customizable but also sufficiently predefined at the same time, so that configuring them further is only an

[8] Cascading Style Sheets, a styling language describing the presentation of markup documents such as HTML web pages.

[9] https://www.openstreetmap.org/

option and not a necessity. All of these SVG[10]-based visualizations are built around a list of objects, whose elements only consist of *key* and *data* properties.

2.3 Architecture

The software platform's architecture can be divided into three well separated components. The bottom layer, a document-oriented MongoDB database, is responsible for the permanent data storage. This gets queried by a backend server implemented in Golang, where the retrieved documents are first encoded into intact models, then these are further transformed into more compact transfer objects specific to the request. These requests are received at the server through a RESTful[11] API[12] [36], and are originating from the web client implemented in TypeScript, built on the React web framework.

Within the multi-layered architecture the presentation layer is implemented in React. The application can be also categorized as a Single Page Application (SPA), meaning that the view layer is fully built up in the frontend part [5]. Even the business logic resides predominantly here, while the backend server's only tasks are about providing and managing data.

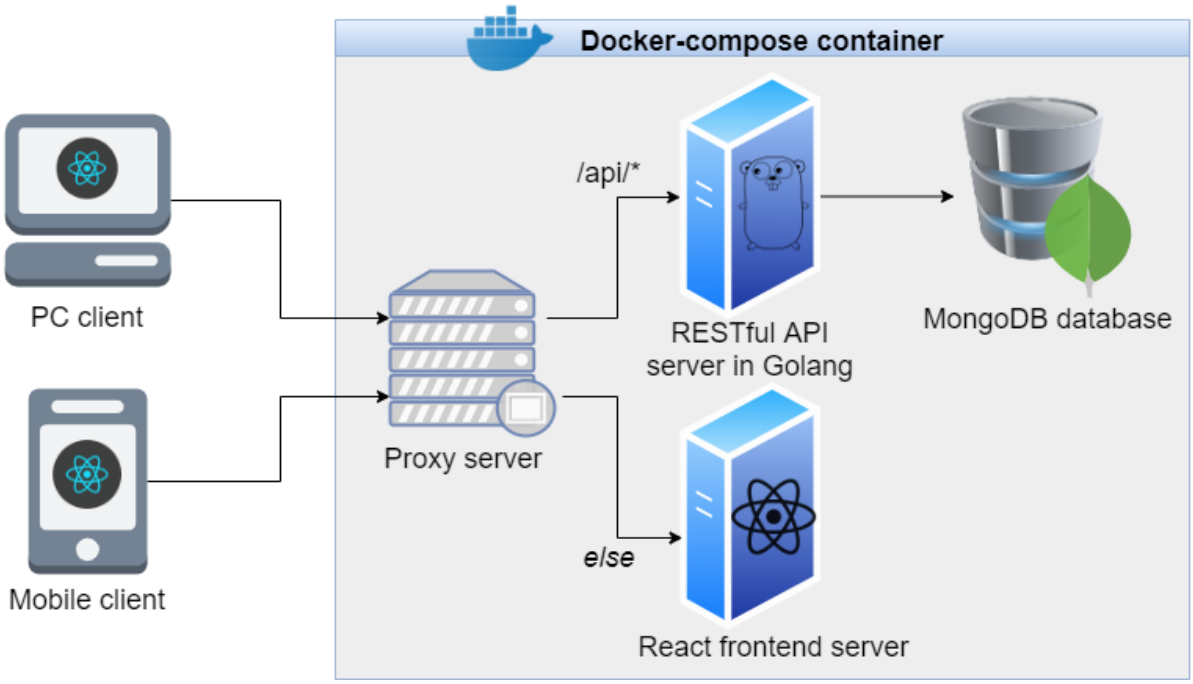

Figure 2.1: Overview of the deployed platform's architecture.

[10]Scalable Vector Graphics, an XML-based 2D vector image format.
[11]Representational State Transfer, a software architectural style defining conditions on the usage of HTTP.
[12]Application Programming Interface, a set of definitions and protocols defining interactions between software components.

The application is deployed in a multi-container environment, as represented on Figure 2.1. The three components are communicating via an internal network, while a fourth container - incorporating a proxy server - exposes the platform to the outside world. It also distributes incoming requests from various devices' browsers between the web client and the backend server. The former only serves static files and is only accessed on the first visit, since being an SPA, the entire webpage is transmitted to the user's device. The Golang server replies to the RESTful API requests starting with the pattern */api/**, providing data access and data manipulation services. More about the deployment is discussed in Section 4.2.

Chapter 3

Implementation details

Now that the underlying technologies were introduced, this chapter focuses on some implementation details about the more interesting functionalities of both the backend and the frontend applications.

3.1 The backend server's implementation

The section first presents the steps of the data processing, then explains the detailed architecture of the backend server. The last subsections demonstrate the solutions for implementing security and for automating e-mail notifications.

3.1.1 Problems of the data processing

The provided database which served as the base and motivation of the project was initially in Access[1]. It was very poorly maintained, exhibiting significant differences between the priest records regarding available data quantity and quality.

Due to these reasons, the data had to be transformed into a more usable format. The first step was converting the single-table Access file into a relational database, in order to categorize the data and better understand how it is built up. Then the *Priests* and *Parishes* MongoDB collections were built up from these tables.

The original database only contained explicit data about priests, the collection of parishes was generated by an algorithm that aggregated data from priests' placements. Consequently, the

[1] Access is Microsoft Office's graphical database management system.

production database only contains the parishes' name and the priest list, but no other fields such as history or origin, which will be filled in by the Protestant Institute's librarians.

This algorithmic data propagation in some cases created multiple parishes with very similar names, due to the misspells present in the original database. The team only employed algorithms for the most obvious of these cases. However, to prevent potential data loss, most of the duplicates could not be algorithmically cleaned out, since many village names only differ in one character from one another, bigger towns often accommodate multiple parishes and some of the original data fields contain additional information about the priest. Ultimately, the merging of the duplicate parishes and the clarification of the priests' biographical data will also be done by the Protestant Institute's librarians.

Most of the parishes have coordinates, which were retrieved and inserted by Python scripts. These coordinates were queried by the parishes' names using Google's Places API [31] with a language preference and a location bias favoring hits close to Transylvania. The aforementioned duplicate parish entities have the exact same coordinates, thus their markers are placed on top of each other in the map view. The Leaflet plugin mentioned in Section 2.2 deals with this problem.

3.1.2 Data access

One particular property of the document-oriented data model used by MongoDB, which fits the requirements of the application, is that it does not require a field and type defining schema. By not having to create and enforce a schema as it would be necessary in case of SQL databases, it was possible to keep all the provided data fields without having performance drawbacks caused by the large number of missing or invalid data.

In the project there are five different collections defined. The *Priests* collection contains the information associated with priests, potentially even 150 different data fields per record, which can be categorized into general data, family, qualifications, placements occupations, literary works and references, disciplinary matters, pictures and files.

The *Parishes* collection stores data related to parishes, together with the list of priest who ever served there. In the *Users* repository the user accounts of the system can be found. The fields include the name, the e-mail address, the hashed password and the access level of the user. The last two repositories store data about the tokens used for the validation of password changes and about the news displayed on the public main page of the application.

```
parishObjectID, err := primitive.ObjectIDFromHex(parishID)
if err != nil {
        logrus.WithField("error", err.Error()).Error("ObjectIDFromHex error")
        return err
}
opts := options.Update().SetArrayFilters(options.ArrayFilters{
        Filters: []interface{}{bson.M{"elem.placeID": parishObjectID}},
})
filter := bson.M{"mainPriestPlaces": bson.M{"$exists": true}}
change := bson.M{"$set": bson.M{"mainPriestPlaces.$[elem].place": parishName}}
result, err := impl.collection.UpdateMany(ctx, filter, change, opts)
```

Listing 3.1: Updating placement location names in the priest documents.

In the Golang server, connection to this database is realized by a well-separated layer of structures, one for each collection of entities. The database queries are constructed in a similar manner to the native MongoDB commands, relying on nested structures to express complex conditions and actions. An example from the priest store's *UpdatePlacementLocation()* function can be found on Listing 3.1. The function, as its name suggests, updates the location name of a placement in all of the priest documents containing it, after the respective parish was renamed. The necessity for this operation is caused by a trade-off, where the team chose to have the data redundancy of storing the parish name as placement location in the priest's document as well. The reasoning is that the priests' with their placements are queried orders of magnitudes more often than the event of a parish having its name changed.

Nevertheless, the code snippet starts with the transformation of the string identifier to a native one, followed by the Golang-specific oversimplified error checking. The identifier conversion is an often repeated operation in the different repository functions, since the other layers of the Golang server do not contain any code specific to MongoDB, and the access to these store functions is only possible through a set of general interfaces. This means that the data access layer is independent and interchangeable within the application.

In the following, an *options* variable is created, which together with the *elem* array index variable defined in the penultimate row, identifies all placements in the placement arrays of the priest document with the given identifier and sets their *place* field to the new value. The *filter* created before the *change* variable helps avoiding indexing errors in non-existing lists. The number of updated documents can be extracted from the first of the two returned values of the *UpdateMany()* call.

Due to the frequent reinitialization of the production database motivated in the previous

subsection, the backend server also communicates with a MySQL database[2]. These importing functions are only ran at server startup and they only use the MongoDB interfaces' *deleteAll()* and *insertMany()* functions to batch process the data. This database is not part of the end product, it is only used in development.

3.1.3 Data Transfer Objects

The Data Transfer Object (DTO) is a software design pattern which is mostly used when multiple standalone applications interchange large amounts of data. In the backend server, both the incoming and outgoing JSON messages are structured in different DTOs, meaning a high level of abstraction is realized between the internal data model of the MongoDB collections and the entities requested by the frontend client.

The conversion between these two representation is done by so-called assemblers. These are simple functions utilizing the constructors of the target structures to transform the required data from the source entities.

The design pattern's biggest advantage comes from the abstraction of the data, since different parts of the web application require different fields of the entities stored in the database [8]. For example, when displaying search results, only the important dates and the placement list of a priest is used, while the profile page needs almost all of the available data. The chart creator page uses another subset, only years instead of full dates and the size of some structures instead of their content.

Besides the field selection, type conversions can also occur. Some data types of the backend, such as the identifiers specific to the database acquire a JSON-compatible form before being sent out.

The usage of DTOs provides multiple additional benefits. The JSON messages' sizes and quantities are significantly reduced, thus the communication becomes faster and more reliable. At the same, they streamline the development of the web client, by preselecting the required data for the components. The security aspects of DTOs will be undertaken in Subsection 3.1.5.

3.1.4 Detailed backend architecture

Since the application logic mostly resides on the web client, as discussed in Section 2.3, there was no need for a separate service layer between the RESTful API (i.e. controller) layer

[2]MySQL is an open-source relational database management system.

```
parishes := myRouter.router.Group("/api")
{
        parishes.PUT(
            "/parishes/:id",
            auth.Authentication(auth.EDITOR, auth.ADMIN, auth.OWNER),
            myRouter.parishCtrl.Update
        )
    [...]
}
```

Listing 3.2: The router's usage.

and the previously presented store component. Certain operations, such as the data aggregation for the chart designer or the sorting of various arrays are implemented directly in the web client, others take place in the controller's functions.

The controller itself is simply a collection of methods handling the incoming RESTful requests. The *Gin* router package implements the pairing between the URIs[3] of the API's endpoints and the handler functions. In Listing 3.2 the syntax of one such assignment is shown, where the *"/api/parishes/:id"* address is paired to a function of the *parishCtrl* controller struct, and only authorized for signed in users with an access level of *Editor* or higher.

The handler functions are very similar in structure. First, a JSON decoder extracts the body and the parameters of the request from a context variable. Then the content is transformed into either DTOs or directly into the backend's internal model. The next steps are invoking the database functions and executing the custom logic of the functionality which usually involves the usage of the assemblers. Finally, the result is sent back to the web client, together with one of the following status codes, which is also utilized there: 200 - "OK", 204 - "No content", 400 - "Bad request", 401 - "Unauthorized", 409 - "Conflict", 500 - "Internal error" etc. The result itself can range from a single record or a list of records in case of a GET request to the identifiers

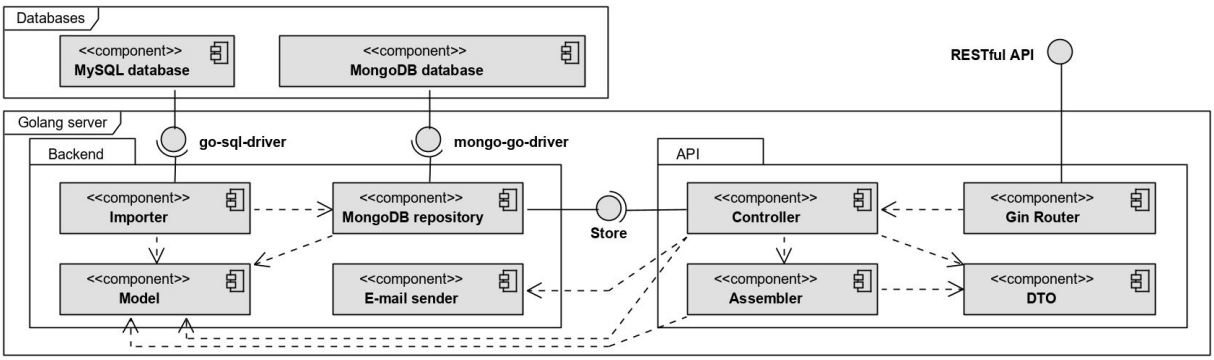

Figure 3.1: Components of the backend server and their interaction.

[3]Uniform Resource Identifier, unique paths in this case.

of the recently created or updated entities for responding to PUT or POST requests.

The architecture detailed in the last three subsections is visualized on a component diagram in Figure 3.1.

3.1.5 Security and authorization

The security of the application was a key aspect throughout the whole development process. Aside from the mechanisms applied in the web client, described in Subsection 3.2.3, the backend server also relies on multiple different methods in order to realize data protection.

Such a method is present in the previously explained Listing 3.2, where the authenticator interceptor filters out the unauthenticated senders, and it can further restrict by user roles as well if additional parameters are provided. This is realized by attaching the session cookie to the HTTP request, from where the server-side session and its parameters can be retrieved.

The DTOs investigated in Subsection 3.1.3 offer another level of security for the platform. Due to them, users with the *Guest* rank cannot read the priest's personal data that is not present on the search result cards, not even directly from the API, since their requests are not able to receive these kinds of information. This field-selection also encompasses leaving out internal identifiers and timestamps from DTOs, whose transmission to the frontend client could cause potential security holes.

Password protection can be broken down into multiple levels. First of all, they cannot leave the backend server under any circumstances, since they are not part of any outgoing DTO. Second, the deployed application is containerized and the *proxy server* communicating with the outside world only connects to the backend and frontend servers, meaning that accessing the database is impossible. Additionally, passwords are encrypted and salted multiple times during registration, using the *bcrypt* base library.

3.1.6 Automatic e-mail notifications

The application relies on Google's SMTP[4] server for sending out notification e-mails [1]. The service's free tier limits sent e-mails to 500 a day, however the Protestant Institute's assumption about the magnitude of the user base fits in this limit.

When a new registration request is submitted, all admins of the website receive an e-mail with the new user's data and introduction. After a pending account is accepted or rejected,

[4]Simple Mail Transfer Protocol, a set of constraints for e-mail transmission.

the user receives an e-mail about the decision, again accompanied by a personal message. Aside form this, e-mails are also sent when changing passwords, containing a hyperlink to the password resetting form that is valid for 48 hours. These dynamic data fields, such as user names and personal messages are inserted into the HTML e-mail templates by the *html/template* package.

Because depending on an external service always carries some uncertainty with itself (however, without an SMTP server all e-mails would most probably be marked as spam), the server continuously monitors its situation, by saving the last returned values. Whenever an error occurs in e-mail sending, the admin dashboard web page will show a notification stating that the maintainers of the application should be contacted. This error disappears when new e-mails can be successfully sent again.

3.2 Implementation details of the web client

After SPA-oriented frontend frameworks such as React started to emerge in popularity, the development complexity has gradually shifted from the backend server towards the web client [3]. This section presents some of these newfound challenges and their solutions.

3.2.1 React state management without a Store

In case of larger applications, the *react state* explained in Subsection 2.2.1 often leads to dividing the data between different components. To counter this, libraries with the purpose of collecting the program state into global store(s) quickly appeared after React became open source. Packages such as *Redux* and *MobX* made the codebase easier to oversee, but due to their complex and verbose syntax, they also significantly increased its size and complexity [10].

Another problem derives from the fact that React components' behavior is programmed in so-called lifecycle methods, which collect into one function all the actions of a component's elements that should be performed when a specific event occurs. Such functions are the *componentDidUpdate()* invoked after a refresh, the *componentDidMount()* triggered when inserting a component into the DOM, and the *componentWillUnMount()* executed when the component is about to disappear. This logic leads to another instance of code scattering. In this case the different events related to one element get lost between multiple lifecycle methods.

These problems are attempted to be addressed by the *Hooks API*. The update rolled out not

```
useEffect(() => {
    if (response) {
        setShowLoader(false);
    }
}, [response]);
```

Listing 3.3: The *useEffect()* hook's syntax.

long before the project launched, thus apart from the official documentation [39] few external resources were available [17].

Hooks are functions, by which the component can "hook" into the component lifeycle. They offer solutions to both of the mentioned shortcomings. First of all, state management is simplified, the global store-based libraries are replaceable by *useState()* or *useReducer()* calls [10] operating on immutable variables via pure functions. The usage of these hooks makes the architecture of the frontend component inherently simple, free of additional layers. These components are said to be self-contained and self-aware, since they own all the data they rely on and contain all the functions which can affect them. The *useState()* hook attaches an immutable variable and a function that can modify it to the state. The

```
const [filters, setFilters] = useState(initFilters);
```

row, for example, binds the filters' value to the state, providing the *initFilters* object as an initial value.

Since hooks can only be used in functional components, React classes are completely replaced by these. Therefore, events related to the same element are not scattered in different lifecycle methods, but are grouped together by the *useEffect()* hook which is executed when the assigned data changes. In the last line of Listing 3.3, the change of the value(s) given in square brackets trigger the function inside the hook. In this case, when the *response* object has arrived from the backend server, the loader component gets immediately disabled. For this to work, both the *response* and *showLoader* variables need to be part of the program state, via *useState()* calls.

Developers are also able to create their own hooks, making special logic reusable between multiple components. An example of this is the *useDelete()* generic function (see Listing 3.4). Here, first the generic response and the variable showing the error is attached to the state. Then an asynchronous function is declared for sending DELETE calls through the *Axios* library [22]. The interceptors of this asynchronous function modify the two stateful variables according to the response. The hook can be used by the

```
const useDelete = <T extends {}>(): { response: T | undefined;
showError: boolean; asyncDelete: (url: string) => Promise<void> } => {
    const [response, setResponse] = useState<T | undefined>();
    const [showError, setShowError] = useState(false);
    const asyncDelete = async (url: string) => {
        const URL = process.env.REACT_APP_API_URL + url;
        await axios
            .delete<T, AxiosResponse<T>>(URL, { withCredentials: true })
            .then(
                (resp) => { setResponse(resp.data); },
                (error) => {
                    if (error.response) { setResponse(error.response.data); }
                    else { setShowError(true); }
                }
            );
    };

    return { response, showError, asyncDelete };
};
```

Listing 3.4: The *useDelete()* custom hook.

```
const { response, showError, asyncDelete } = useDelete<ILabelResponse>();
```

syntax, where the first two returned values are ready to be used in the component's logic, while the asynchronous *asyncDelete()* function reference can be utilized for example in the handler function of a *Delete* button.

The hook responsible for executing GET requests works in a similar manner, with two notable differences. It contains an additional Boolean variable indicating if the request is still underway, this is used for showing and hiding the loading animations. Secondly, the asynchronous function is not returned, instead it is part of another *useEffect()* hook which triggers if the URL changes. When using the searching fields on the webpage, a chain reaction happens, where first the URL is modified by the filter changes, then the URL change triggers this inner *useEffect()* hook, sending the request which overrides the old values returned earlier, updating the search results in real time.

3.2.2 Responsive and clear UI

Creating an easy to use and intuitive user interface was one of the platform's key requirement. The discipline of *UX design* applies multiple strategies to make the process of planning the user interface become more procedural, even if it seems subjective at first.

In the beginning of the project, after a concise study of the field's literature [15, 2], the team created multiple design plans for all components of the future website using a mockup creator

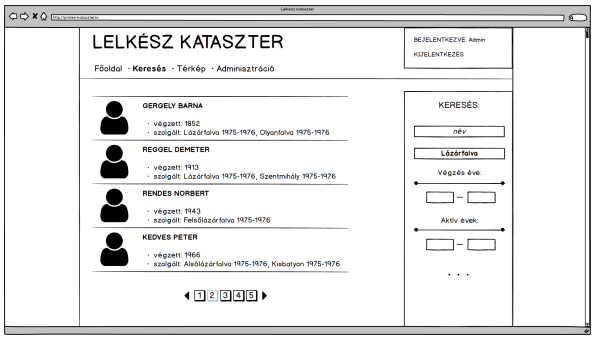

Figure 3.2: Design plan about the priests search page.

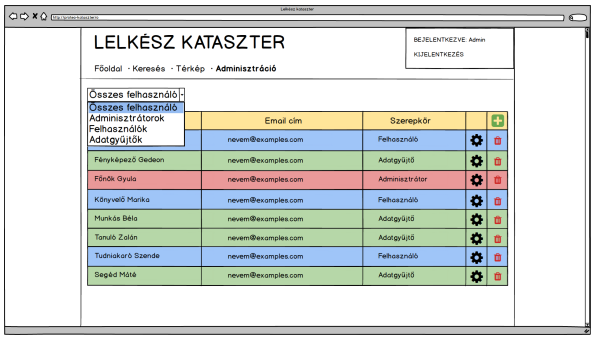

Figure 3.3: Mockup of the administrator dashboard.

and presenter software called *Balsamiq* [24]. The selection between these was made based on the remarks of the client and on further UX research. Two such mockups are present on Figures 3.2 and 3.3, which only differ in the sidebar's placement from the screenshots of Chapter 5. Furthermore, the team also got feedback from a UX specialist during the actual development.

The software platform does not have an Android or iOS application. The team's and the client's common conclusion was that it would have been necessary only if it relied on some native services of smartphones, such as camera integration or location access. Instead, the user interface was designed in such a way that its arrangement fits any screen size, from smartphones to tablets, notebooks and widescreens. For examples please refer to Chapter 5.

The responsive interface and the underlying design was mostly realized using the React-integrated version of the *Semantic UI* (SUIR) package. Every visual item of the web page is either a redefined and themed HTML element from this library or a component pieced together from these. The components' size, color and other visual attributes can all be configured through optional constructor parameters, but in multiple cases these were overridden by manually given CSS commands, so they complement the overall look better. The elements of the library are all fluid, meaning that their size depends on the window's size, while they occupy a near constant percentage of the available space.

There were multiple other methods for realizing the responsivity of the webpage. For example, the navigation bar has two implementations, one containing only the menu options' icons without the labels. Between rendering these two components the decision is made by SUIR's *computer/tablet only* selectors. When browsing on a tablet, the different cards of the search and profile pages automatically stack into one column instead of the usual two, by the *Grid* layout definer element's *stackable* parameter. The arrangement of the components changes on mobile screens too, the main content is placed below the auxiliary controls otherwise present

```
const [filters, setFilters] = useState(initFilters);
const filterURL = Object.entries(filters).reduce((prev: string, entry: [string, string]) => {
    if (
        entry[1] === "" ||
        (entry[0].includes("Start") && entry[1] === filterStartYear.toString()) ||
        (entry[0].includes("End") && entry[1] === filterEndYear.toString())
    ) {
        return prev;
    } else {
        return prev + entry[0] + "=" + entry[1] + "&";
    }
}, "");
const { response, loading, showError } = useFetchData<IPriestMin[]>(`/priests?${filterURL}`);
return (<>
            <Redirect to={`/priests/search?${filterURL}pageNr=${pageNr}`} />
            // ...
```

Listing 3.5: Quality of life features coming form the filtering URL.

on the left side of the pages, while cards still show up in one column. These are achieved by taking into consideration the current resolution of the user's device using the

```
@media only screen and (max-width: 768px) {...}
```

and respectively *min-width* CSS media queries.

One essential difference between UX and UI (User Interface) design is that the former is not only concerned with the visual components of the page, but also considers the usability and the convenience of the functionalities. There are multiple small comfort features present in the application, such as the reset buttons near the double sliders, the real-time search results, or the filter values' and page number's insertion into the current URL. The latter is implemented so that specific searches and results can be shared via the hyperlink. This and the real time results are both based on a specific JavaScript feature: object fields can be iterated over. Listing 3.5 shows how the *filters* object is reduced into a string containing all the search fields that are not in their default states. This triggers a state change, so a new request is sent to the server. Additionally, on the last row of the code snippet, just before rendering the component, the page navigates to the changed URL. This would cause an infinite refresh cycle in most frameworks, but React only refreshes the components affected by a state change, and here only the browser URL is modified.

To fine-tune the interface's font types and colors, the SUIR package was recompiled using a library called *craco*, so that the internal configuration files of SUIR became accessible [45].

3.2.3 Authentication and Authorization

Besides the backend's security measures presented earlier in this chapter, the web client also applies multiple precautions for securing data against unauthenticated access and against accounts with insufficient privileges.

From these, the easiest to notice is that the navigation bar keeps the unavailable paths hidden, depending on user sessions and roles. Similarly, the data manipulation controls are hidden as well, when they are not applicable, such as the *Edit* button on the profiles or the article posting, editing and removing icons on the main page. This way, the unauthorized functionalities are not even present on the rendered webpages.

The custom *GuardedRoute* functional component built around the *react-router* [41] provided *Route* component takes this a step further. The navigation paths listed in a *Switch* component at the application's root are wrapped in this component, as shown in Listing 3.6. The custom function receives the variables necessary for the *Route*, namely the *path* and *render* values, but also the *user* object - encapsulating the data of the logged in user - and a list of permitted user roles. By the former two, it can be determined if the *Route* component should refer to the given path or to an error page. Therefore, unauthorized pages become inaccessible even in the possession of the URL leading to them.

After authentication, the logged in user's session is stored in the browser, until signing out manually. This comfort feature is implemented by using a custom hook responsible for session storage, along with the user object's stateful handling.

It is not necessarily a security measure, but the registration and login forms feature dynamic

```
<Switch>
    <Route exact path="/" render={() => <HomePage user={user} />} />
    <GuardedRoute
        path="/priests/profile/:id"
        render={({ match }) =>
            <PriestProfilePage
                priestID={match.params.id}
                userRole={user ? user.role : undefined}
            />
        }
        user={user}
        roles={[UserRole.READER, UserRole.EDITOR, UserRole.ADMIN, UserRole.OWNER]}
    />
    [...]
</Swicth>
```

Listing 3.6: The usage of the *GuardedRoute* custom component.

error messages, based on a list containing different translation keys. These are collected by monitoring all possible error sources: missing fields, invalid data and error codes received from the backend server's response.

3.2.4 Displaying and managing dates

The correct management of dates in the application was essential, since the searching components heavily rely on them. In the original database these were saved as strings, thus unsurprisingly, contained a vast number of misspells ("19885.04.28"), ambiguities ("1992 Jan first half") and improper extra information ("1919.10 - assistant teacher").

In order to normalize them, a new structure was created, which stores the year, month, day and original string values in different fields. The transformation of the data is performed by an algorithm exploiting multiple regular expressions, and is ran when the production MongoDB database is generated from the MySQL one on the backend. The first step attempts to recognize all three numbers in the date string, and if it is unsuccessful, the year and month or lastly the year is tried to be matched. This method proved successful, recognizing about 95% of year values in the dates.

The presentation of these dates on the web client is performed by two components and two auxiliary functions. Their purpose is that the custom logic involved in managing the dates is easily reusable across all parts of the website.

The *DateInput* component consists of four inputs for the four fields (the last one being the legacy string or a comment), and of a conditionally appearing error message. The latter appears when there is a problem in the edited date, be that a nonexistent month and day combination or even a leap year related issue. This error message is also present below the controls used for canceling and committing the modifications made in the profile's editing mode, and if any error is present, committing the changes is not possible.

The *DateDisplay* component aims to present the dates as accurately as possible. If all the numeric values are present, a normal full date string is returned. In case of missing day or month values a partial date is still returned. When mousing over dates, the original string value form the initial database is also shown in a popup text-bubble. If the year could not be retrieved, this original string is shown instead of the date. Finally, if that is also empty, a *No data* message appears.

3.2.5 Translation

Texts shown on the website are possible to be rendered in multiple languages through the *react-intl* library [26]. The English and Hungarian texts are stored as key-value pairs residing in two JSON files. Access to these translations in the components is realized by integrating them into the application through instantiating the *IntlProvider* component as the outermost layer of the application.

The language selected in the navigation bar drop-down is inserted into this root component through a custom hook, which is also responsible for the local storage of the last selected language, thus the user's preference is saved between visits.

The circa 350 pieces of translated texts can be displayed by the *FormattedMessage* component or the function returned by the *useIntl()* hook. The latter function returns a string, which remains usable when a text should be given as a constructor argument to another component. This helps in integrating translated labels into premade elements of the SUIR library, and is also used when data input/output fields on webpages are not hardcoded, but inserted by iterating through an entity's attributes.

It is worth mentioning, that these two components can be wrapped too. There are three such functions in the project, which transform the translated texts into bold, italic or underlined versions of them, and can also apply other optional styles to them. Components built around such minimal changes help in keeping the more complex parts of the code readable.

3.2.6 The custom chart designer

The data visualization is implemented with the *Reaviz* package [42]. Its most prominent advantage, from the point of view of the project, is that it utilizes the same data model for all of its available charts. Therefore, data aggregation can be realized in a simple and concise manner, independent of the chart's type. Another advantage of the library is that it is opinionated, meaning that most of its design choices are made internally, therefore the theming's data-specific implementation is not falling on the developer.

The data processing is implemented on the frontend, with the combination of multiple functional operations such as *forEach*, *reduce*, *filter* and *sort*. The underlying data model received from the backend server contains only the ready to use values, e.g. years instead of full dates and only the magnitudes of the different data sections. The first two steps of this process can be examined in Listing 3.7. The aggregation on the user-selected fields is realized

```
priests.forEach((p: IPriestChartData) => {
    if (p[xAxis] !== undefined && p[xAxis] !== 0 && p[xAxis] !== "") {
        usedDataSize += 1;
        const match = chartData.filter((item: IChartData) => item.key === p[xAxis]);
        if (match.length === 0) {
            chartDatum = { key: p[xAxis], data: p[yAxis], counter: 1 };
            chartData.push(chartDatum);
        } else {
            match[0].data += p[yAxis];
            match[0].counter += 1;
        }
    }
});
// ...
const temp = chartData.filter((item: IChartData) => item.counter < minGroupSize);
chartDatum = {
    key: fm({ id: "charts.etc" }),
    data: temp.reduce((prev: number, next: IChartData) => prev + (next.data || 0), 0),
    counter: temp.reduce((prev: number, next: IChartData) => prev + (next.counter || 0), 0)
};
```

Listing 3.7: First two steps of the chart data aggregation.

by the *xAxis* and *yAxis* strings selected through the *Common property* and respectively the *Value axis* drop-down menus. The filter call verifies whether the current category exists in the already grouped data, and then the found value is incremented or a new category is created.

The listing's second part also presents the unification of small groups, which improves the comprehensibility of the charts. Its presence and its threshold are also configurable by the user. The first row after the empty line sums up the groups smaller than the provided limit, then aggregates these into a new category, which is constructed using multiple *reduce* calls.

In the following steps of the algorithm not shown in the listing, among other things the displayable final values of the diagram are computed using the *data* and the *counter* fields, and the small groups are removed from the data. Finally, sorting is done by value or by name and the array is passed to the Reaviz components so that the chart can be displayed.

Chapter 4

Development tools and methodologies

This chapter presents the development methods and tools used in the project. The first section discusses the exact agile techniques followed during the development, then Section 4.2 explains how continuous integration and deployment is accomplished within the project. The next section focuses on testing, while the last one mentions some external tools that were used.

4.1 Agile project management

At the beginning of the project, the team opted for a widespread agile method, namely Scrum. The principles of incremental and iterative work matched the continuous changing of the development circumstances through the summer internship, the group project phase and the further development during the two semesters. In *Scrum* [43] the development process is divided into intervals of a few weeks length called *sprints*.

Every sprint starts off with a *sprint planning*, where the team members determine what needs to be done and formulate short user stories. After assigning point values to these, based on complexity and client priority, the team chooses as many tasks as they can predictably complete by the end of the sprint. These user stories are added to the so-called *sprint backlog*. During the sprint itself, a short discussion takes place every day to help the development team stay up to date with each other's work and solve problems together. At the end of an iteration, the finished functionalities are presented to the client. This is followed by a retrospective meeting where the team discusses the positive and negative experiences of the past weeks. Figure 4.1 visualizes the interlacing of the daily and the multi-week cycles described above.

During development, the status of each user story was recorded on a virtual *kanban board*,

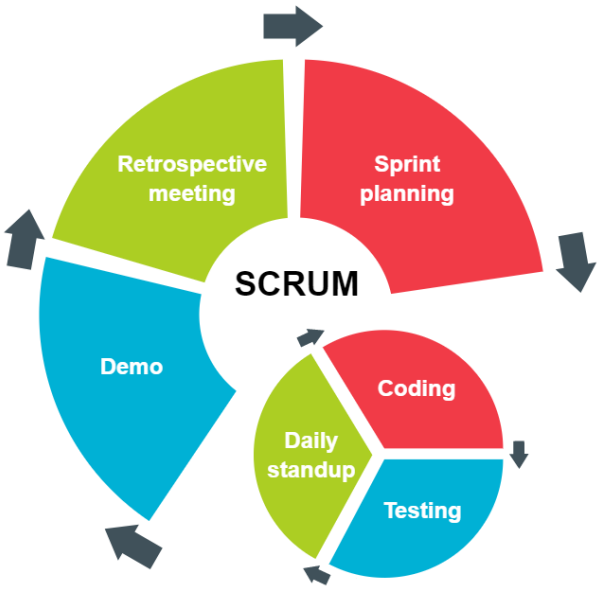

Figure 4.1: Iterations in the Scrum technique.

which allows the team to visually track the progress of various tasks [18]. These tasks appear as cards on the board and move step-by-step through the following columns (i.e. stages):

1. *Backlog*: tasks chosen in the last *sprint planning*.
2. *Doing*: user stories selected by developers, under implementation.
3. *Review*: tasks that are complete and ready to be verified.
4. *Merged*: features that are accepted by the reviewers.
5. *Closed*: stories that are found in the *Merged* column and are accepted by the client, when the sprint is completed.

4.2 Version control, continuous integration and deployment

Version control is realized using Git. The multiple git repositories of the backend, frontend, deployment, UX design and data processing subprojects are all hosted on GitLab.

Continuous Integration and Continuous Deployment (CI/CD) [19] played a major role in the development process. The first three of the aforementioned git repositories are also connected to GitLab pipelines. This allows the automatic execution of a list of interdependent tasks after any change of the codebase is pushed to the remote repositories, thus immediately indicating various kinds of problems in the code or when performing deployment related tasks.

Figure 4.2 shows the structure of the three CI/CD pipelines configured in the project. The first two are part of the frontend and respectively the backend repositories. All of their stages are

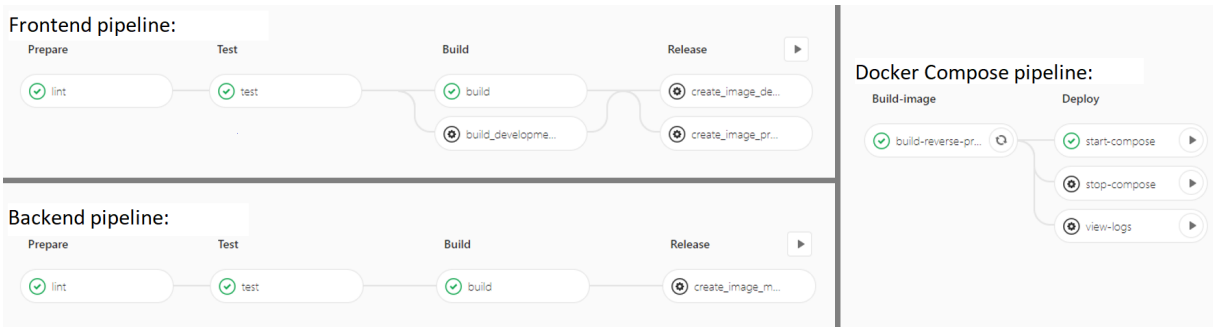

Figure 4.2: The web client, backend and deployment pipelines with their stages and jobs.

similar in purpose. The first of these is the static code analysis carried out by the tools *golang-ci lint* [29] on the backend and *TSLint* [37] on the frontend. The next stage runs the tests and calculates their coverage. More about testing is discussed in the following section. If the tests were successful, the application is compiled within the next stage's *build* job, a short example of which can be found on Listing 4.1. In the code snippet, first a job named *build* is defined and assigned to the stage with the same name, then a folder is created where the Windows and Linux compatible versions of the backend server are placed. The last part declares that this folder is an artifact, thus it is stored for a time in the GitLab servers to be used by the following stages of the pipeline.

Besides these, there is also the fourth and last stage, namely release, implementing *Continuous Deployment*. These jobs are automatically triggered on the master branch - or otherwise can be manually launched - after the build job succeeded and created its artifacts. This plays a role in automating the deployment: when the release process starts, a new *docker image* [4] is created. Listing 4.2 presents the Dockerfile of the backend server, which creates an image of a virtual operating system (from an already existing one) to run the application compiled in Listing 4.1 together with its configuration and other static files. These system images are uploaded to Gitlab's registry from both the backend and the frontend pipelines. From

```
build:
  stage: build
  script:
    - mkdir builds
    - go build -a -o ./builds/kataszter-backend_linux
    - GOOS=windows GOARCH=amd64 go build -a -o ./builds/kataszter-backend_windows-x64.exe
  artifacts:
    paths:
      - ./builds
    expire_in: 1 day
```

Listing 4.1: The build job of the backend server's pipeline.

```
FROM ubuntu:18.04
WORKDIR /kataszter
COPY ./static/ ./static
COPY ./config.yml ./config.yml
COPY ./builds/kataszter-backend_linux ./backend
EXPOSE 8080
CMD ["./backend"]
```

Listing 4.2: Dockerfile of the backend server.

here the third, deployment pipeline can access the completed images to create the production build detailed in Section 2.3 using a tool named *Docker Compose* [25], and deploy it to the test server. The frontend pipeline has two sets of jobs in the last two stages. Since configurations cannot be injected into React applications after compiling them, there are separate jobs for the development and deployment environment's artifacts.

To demonstrate the importance of these pipelines and the time potentially saved by them, it is worth mentioning that they have been run more than 1,100 times during the development of the project so far.

4.3 Verification and testing

Both the backend and frontend parts of the software platform are tested in-depth by automatic methods, most of them integrated into the pipelines discussed in the previous section.

The test job on the backend pipeline only consists of unit tests of the repository layer. Here the coverage is 100%, which is calculated by a shell script, based on the coverage values per package produced by the test runner.

Additionally, the backend server is thoroughly tested by a software called *Postman* [38], which simulates API requests and verifies preconfigured details of the responses received, such as response time, status code and checking the received data itself. There are a total of 260 test cases associated with 87 requests. These include all available API endpoints of the backend. Some of the test cases also attempt to access the endpoints under the circumstances when the request sender is unauthorized or not logged in. A test case verifying the status code and the empty response body for the

```
GET {{server}}/api/priests?name=ilonk
```

request can be seen in Listing 4.3. Currently these API tests can only be executed in a local environment, their inclusion to the automated pipelines with *newman* (a command line version

```
pm.test("response is NoContent", function () {
    pm.response.to.have.status(204);
});
pm.test("response body is null", function () {
    pm.expect(responseBody === null);
});
```

Listing 4.3: Example Postman test case.

of Postman) is an option for further development.

The frontend section is verified by *snapshot testing*, which in essence, ensures that no unexpected changes are made to the user interface. A typical snapshot test renders a configured UI component, then saves it to a file and later on compares it to a component re-created by the same parameters [21]. If there is an unexpected change in the component, the tests will fail. Nonetheless, more complex unit tests can also be designed to verify the mechanisms of handler functions and input fields. In the test cases, the components are isolated, meaning that any outside component and function is *mocked*, i.e. substituted by an object that only mimics its behavior. For example, Listing 4.4 features a test case checking if the *birthTown* input field is correctly wired to the filter's handler function, and also making sure that the layout of the component does not unexpectedly change.

Similarly to the backend, file and module-based coverage is measured here as well, aggregating results from more than 200 unit tests.

```
it("should call setFilters() when the #birthTown input field changes", () => {
    const container = mountWithIntl(
        <SearchPriestsController
            filters={NewTestPriestFilters}
            setFilters={jest.fn()}
            {...handlers}
        />
    );
    const input = container.find("input#birthTown");
    input.simulate("change", { target: { value: "te" } });
    input.simulate("change", { target: { value: "st" } });
    expect(handlers.setFilters).toHaveBeenCalledTimes(2);
    expect(container.toJSON()).toMatchSnapshot();
});
```

Listing 4.4: Example Jest unit test.

4.4 Other tools

Visual Studio Code is a clean and easy-to-configure environment, chosen for both the frontend's and backend's implementation. In addition to the plenty of available third party plugins, it provides syntax highlighting, code completion, built-in command lines, and git and docker integration too [47].

GitKraken provides a visual interface to the git version control system [28], making it easier to execute git commands and switch between different versions, branches or repositories.

In terms of internationalization, an application called **i18n-editor** [14] was used. With a simple visual interface, it allowed the simultaneous modification of different language files required for the translations, thus making the creation and editing of entries more transparent.

The team relied on **Slack** [46] as a communication tool. In the private or group chats messages are possible to be pinned for easily finding them later. Reminders of meetings (and recurring reminders of drinking water and stretching) are simple to set up, and there is also built in code formatting and file sharing.

Figure 4.3 offers an overview of all technologies and tools used, not only from this chapter.

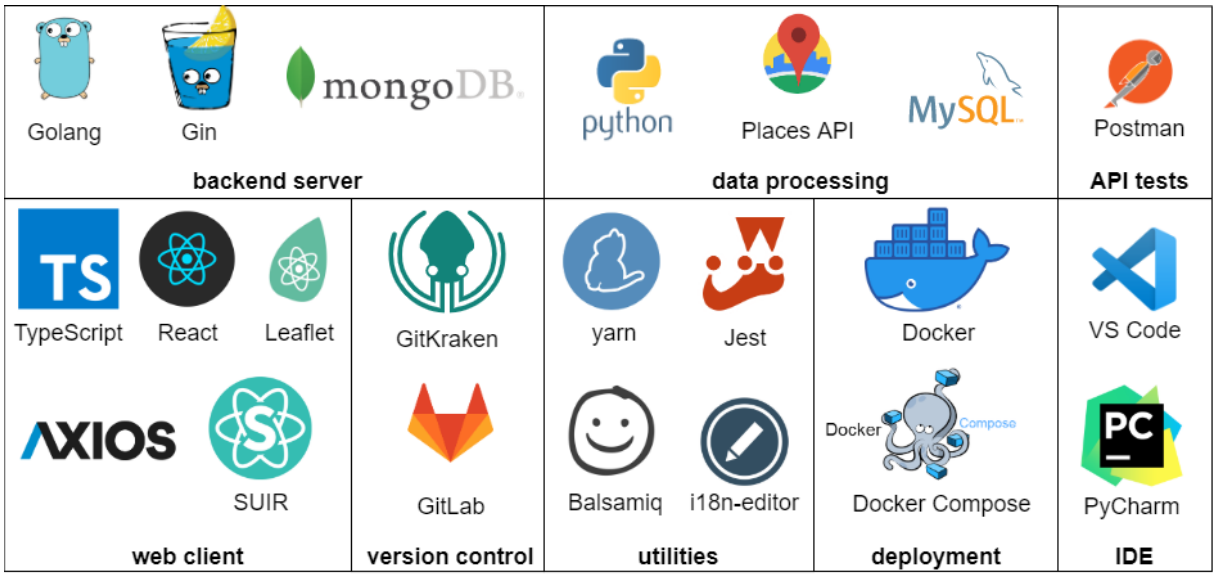

Figure 4.3: Complete overview of all development tools and technologies.

Chapter 5

The usage of the application

The presentation of the user interface is broken down into three sections, the first of which demonstrates all the user account-related functionalities. It is followed by detailing the data-driven functionalities. Finally, the rest of the features will be presented, that were not implemented by the author.

5.1 Authentication and authorization

Before signing in with an account, only the application's homepage, the typical authenticating operations and the language selector drop-down are available from the navigation bar. After logging in, additional functionalities will be accessible, depending on the user's rank.

The registration page (see Fig. 5.2) requires the user to enter their full name, e-mail address, a password twice and a short introduction text. All of these fields are validated in real time, informing the user about possible errors, such as not matching passwords, incorrect e-mail or empty fields. Notably, the introduction field's content serves as a basis for the evaluation performed by the administrators, who receive this introduction and the user's name in a notification e-mail triggered by submitting the registration form.

On the login page (see Fig. 5.1), aside from signing in with an admin-approved account, the component also provides the possibility of requesting a new password by clicking on the *Forgot Password?* label and entering the e-mail address. This generates a password changing link valid for two days, and sends it to the provided e-mail.

Section 1.2 discusses the different user roles and the functionalities associated with them. The features of the website will be presented from the point of view of an admin, so that all

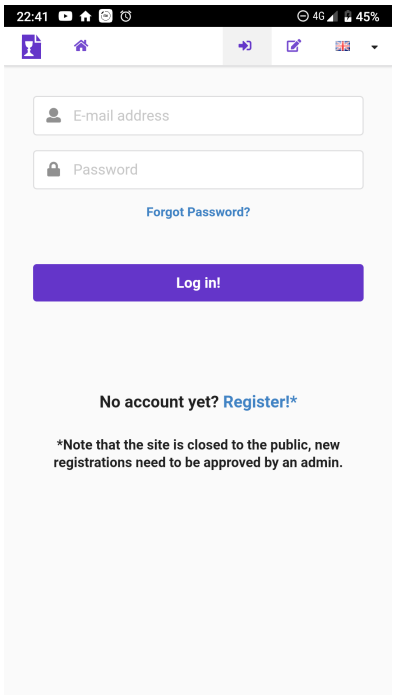

Figure 5.1: The platform's login form.

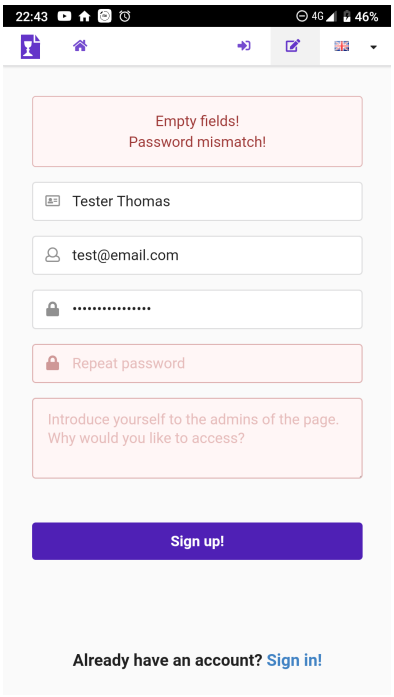

Figure 5.2: Register page with wrong inputs.

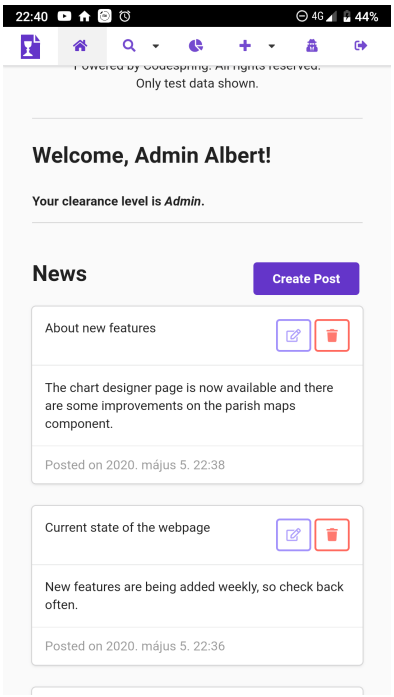

Figure 5.3: Homepage with the news-stream.

of them are present. Users with lower level privileges are able to see only their authorized functionalities in the navigation bar.

The last menu item, if authorized and present, leads to the admin dashboard. As observable on Figure 5.4, different cards show the accounts' details. In addition to the colors associated with the ranks, the sidebar also helps in the navigation between different user groups. Aside from

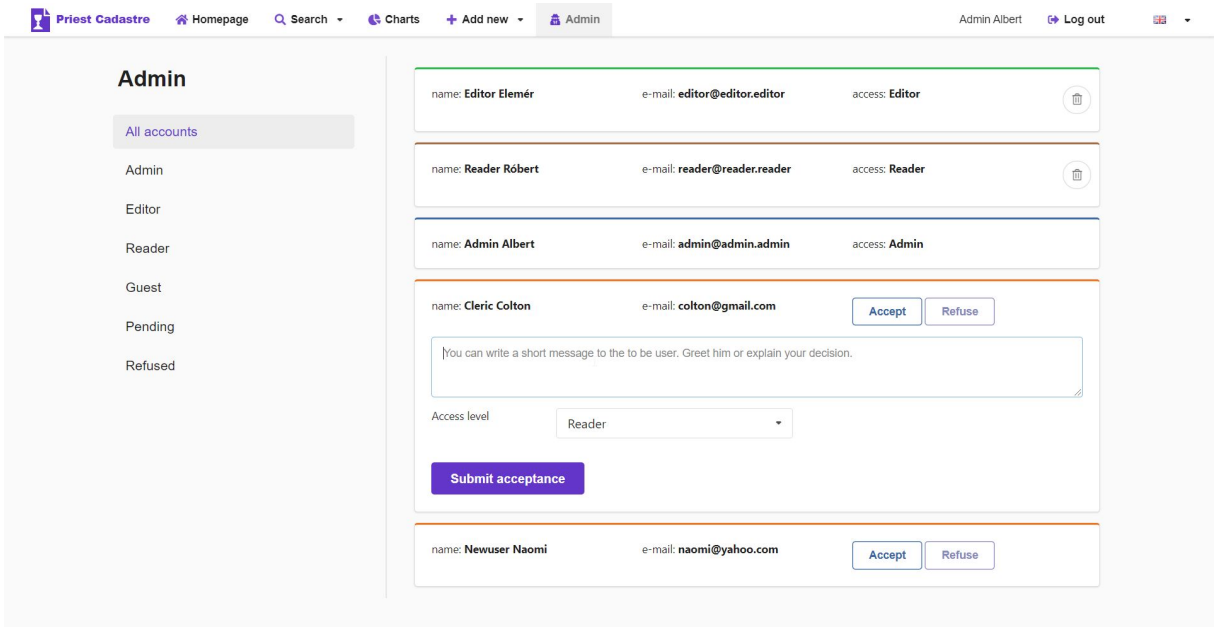

Figure 5.4: Accepting a new account on the admin dashboard.

disabling already existing accounts, the approval or the rejection of the pending access requests is also executed on this dashboard. Accepted accounts also require a user role to be assigned by the administrator. An e-mail is automatically sent to the user in both cases, containing the decision and its reasoning.

5.2 Data-related functionalities

First among the data-driven features of the website, in the *Search* drop-down menu one can choose to filter priests or parishes. The search interface for priests is shown on Figure 5.5. Here, priests can be filtered and target-searched by a broad range of different controls present on the left side of the page. These include text input fields as well as multiple double sliders. While the former can be used to search by specific fields of the priest profile, such as name, birthtown and location of placements, the latter provides adjustable time intervals to narrow down results based on birthdate, years of activity, graduation date and not only. The search results are displayed on the right side of the page, responding in real time to any change of the filters. The list of results is paginated ten by ten, and the controls below the cards can be used to navigate between these pages. By clicking on the cards containing the basic data of a priest, the user is redirected to the priest profile (if the account has the required authorization). The names of the parishes are also clickable on these cards, leading to the parish's profile associated with them.

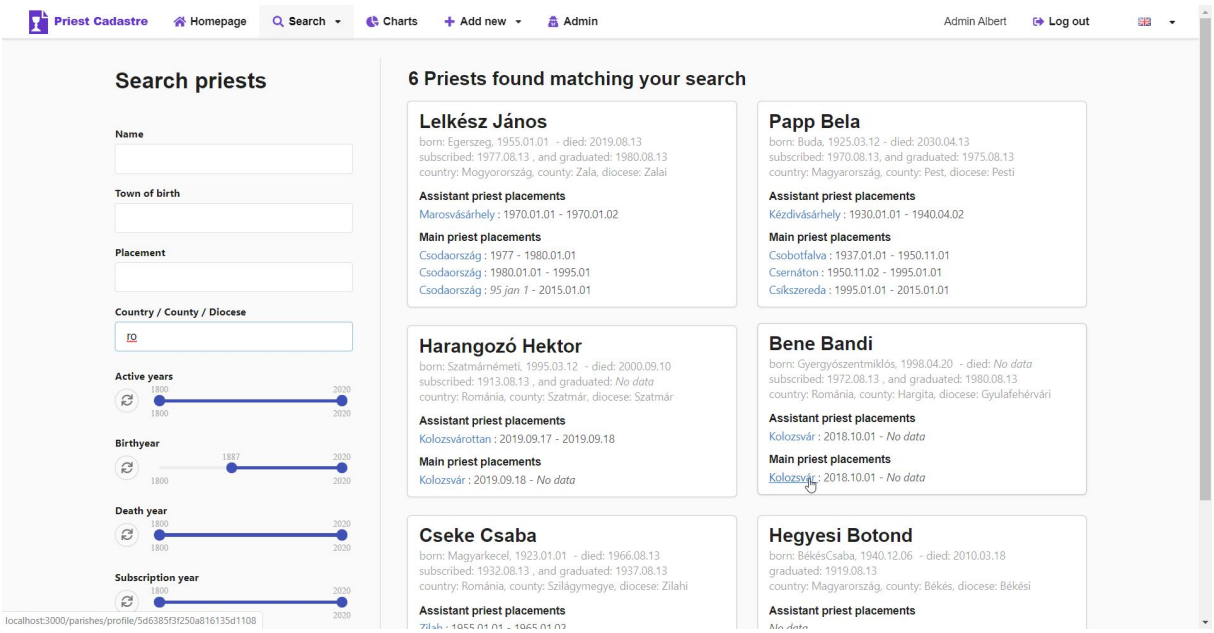

Figure 5.5: Searching priests with results and filtering options.

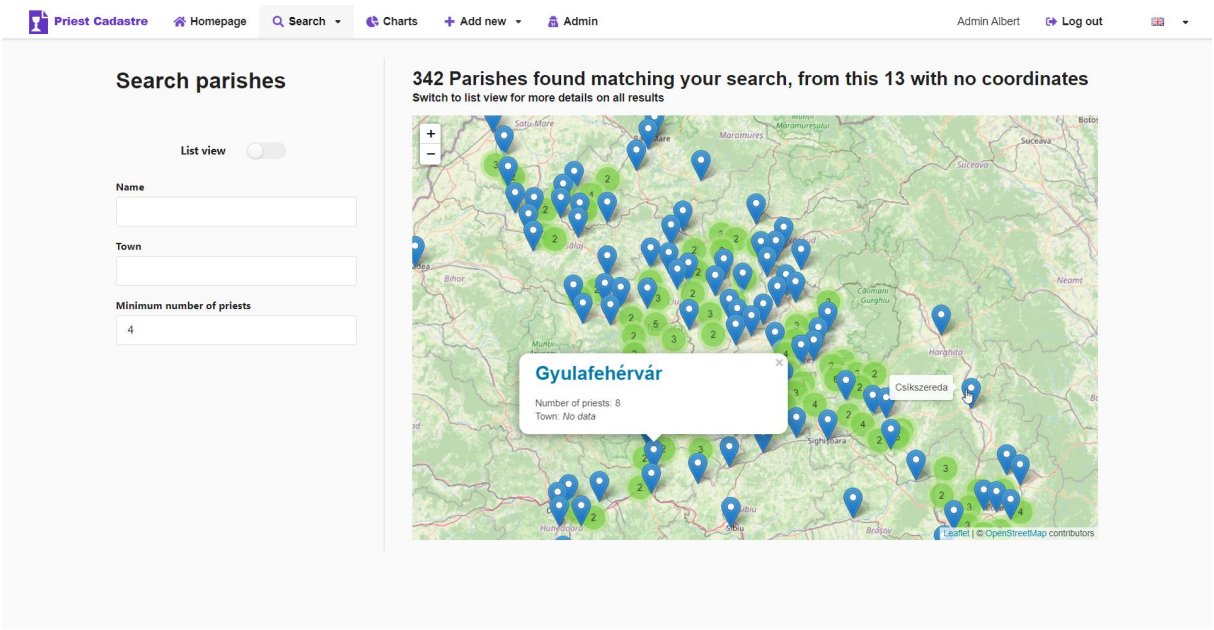

Figure 5.6: Searching parishes on the map.

Searching parishes is also carried out by different filters: its name, location and its minimum number of priests. There are two ways to show the results, the first one being similar to that of priests'. Aside from the list view, the data can also be displayed on a map as well (see Fig. 5.6). Here, parishes close to each other are clustered into a number showing their count, zooming in breaks these clusters down into several markers. Cards in the list and the marker pop-ups can be clicked to navigate to the associated parish profile.

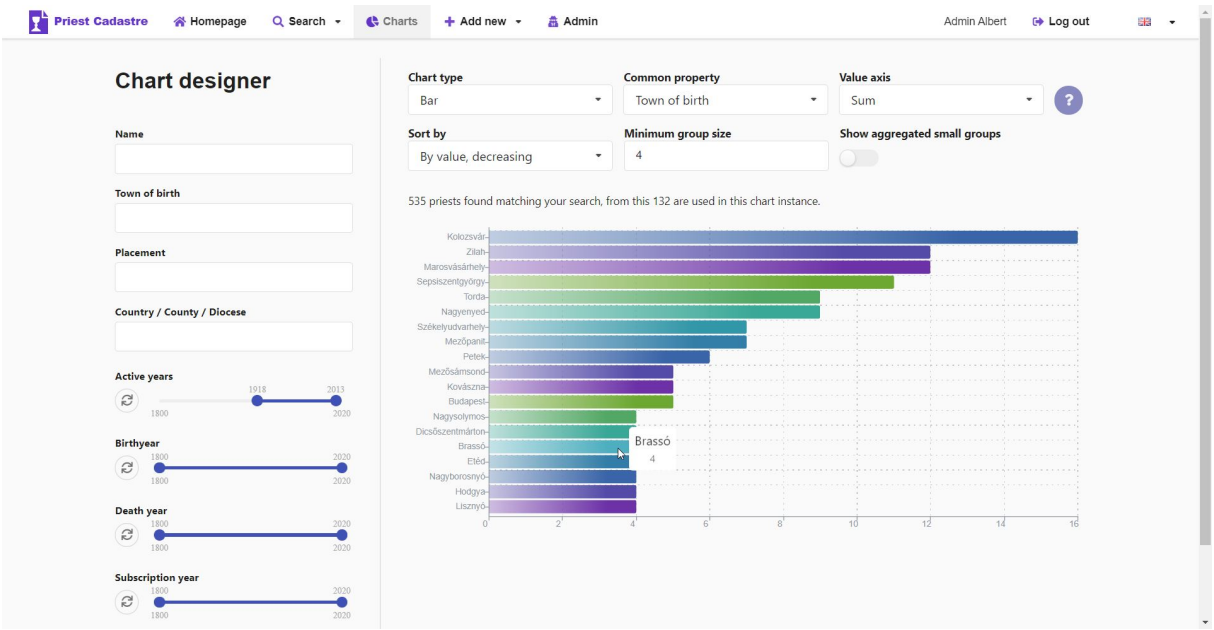

Figure 5.7: Bar chart about the birthplace of priests active between 1918-2013, only groups larger than 4 shown.

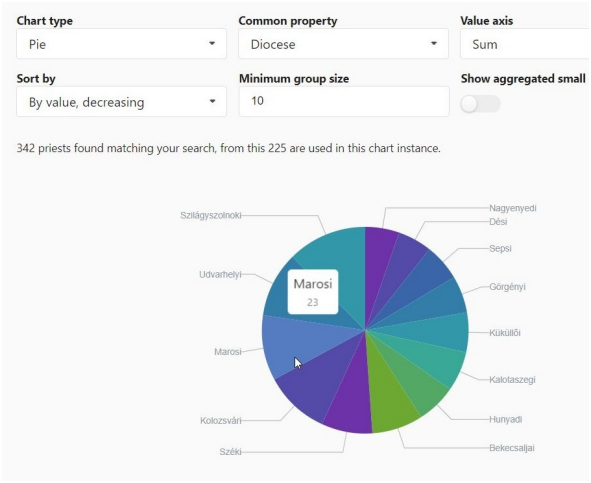

Figure 5.8: Pie chart about the dioceses of Romanian priests, only groups larger than 10 shown.

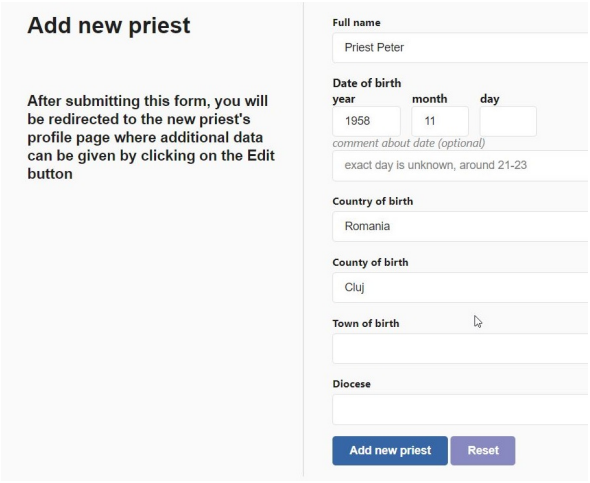

Figure 5.9: Creating a new priest.

The following item in the top navigation bar takes the user to the chart creator. On this page (see Fig. 5.7) a wide range of data visualizations can be easily created and customized through multiple parameters and the aforementioned filtering component. The most important of these options are the two drop-down menus where the displayed data fields can be selected. The first input specifies the main axis, the attribute by which the data will be grouped, such as birthdate or birthplace. In the second drop-down the value associated to these aggregated categories is given. This so-called secondary axis can simply be the group's size, but also can take different averages such as the number of children or of placements. Furthermore, the user can choose the type of the figure: pie, bar or line chart. For the former two, three additional options are available for sorting the data and keeping the visualization clear (i.e. smoothing by aggregating or omitting small groups). Due to the combinability of the multiple options, currently 630 different charts can be generated based on a dataset. Another example of a pie chart is present on Figure 5.8.

5.3 Other features

The section briefly presents the remaining functionalities of the website that were not implemented by the author of the thesis. Not only for the sake of better understanding the accomplishments of the project, but also because the author actively participated in these tasks as well, via pair-programming sessions, code reviews, and in general, when asked by the other colleague.

The homepage accommodates a stream of news (see Fig. 5.3) ordered by recency and a

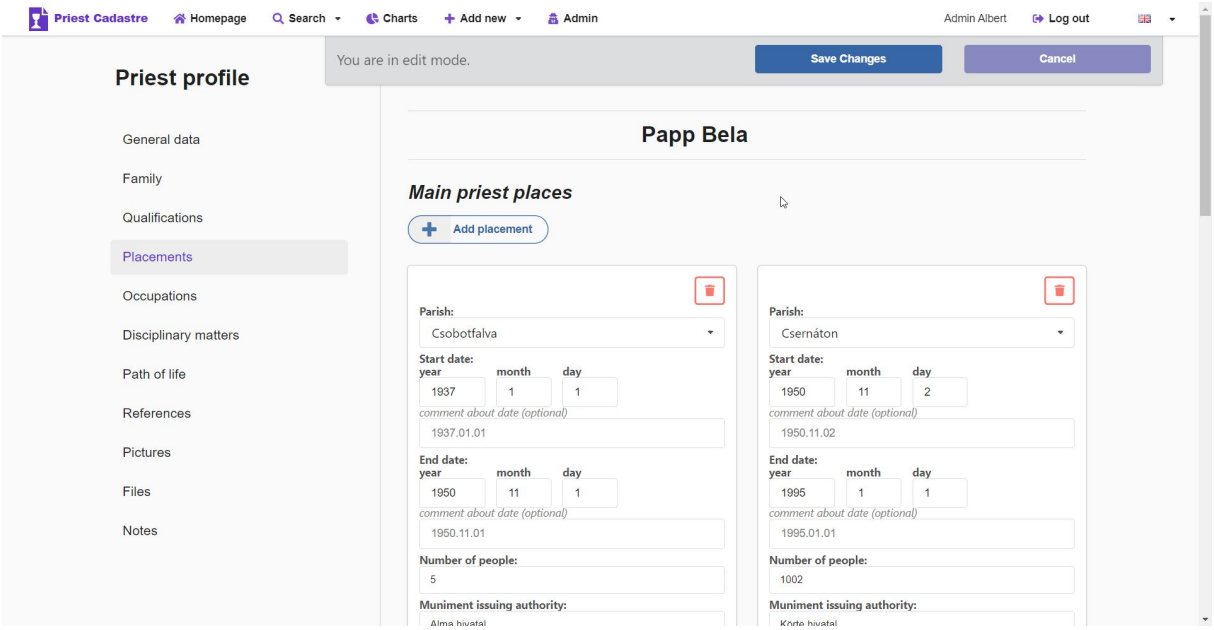

Figure 5.10: Editing a priest's placements.

general description about the project, both of which are readable even without logging in. The *Create Article* button and the icons for editing and deleting these news only appear for signed in Administrators. Clicking on the former two opens a modal window, where the title and the content of the article can be entered or altered.

In the *Add* menu item a drop-down appears, where new priest or parish records can be created by providing some initial information (see Fig. 5.9). Once the form is submitted, the user is redirected to the new profile, where all the other fields can be entered.

There are several navigation links leading to the profile pages within the website. Both on priest (see Fig. 5.11) and parish profiles, different tabs are available in the sidebar, to partition the data into categories such as general data, family, qualifications, placements, occupations, path of life, literary works, files, pictures etc. Notably, the menu item presenting the priest's

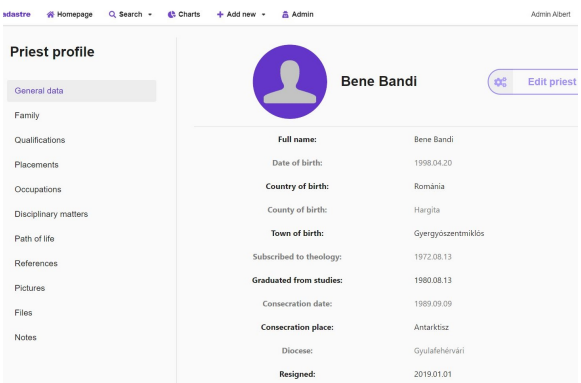

Figure 5.11: Priest profile page.

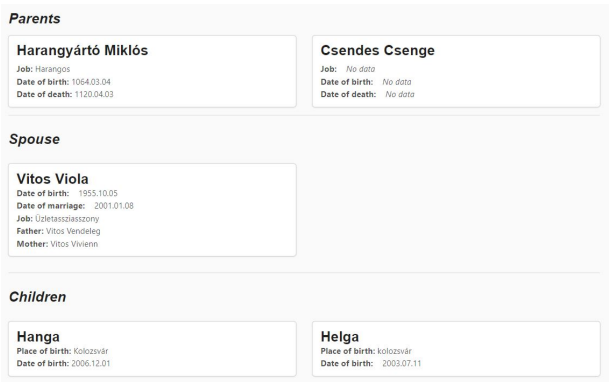

Figure 5.12: Priest's family members.

life path generates a chronological overview of the priest's main life events from the existing data. The parishes' profile page is structured similarly, but with only three subcategories about general data, images and files.

Inside these tabs, information is further grouped. For example, in case of the family subpage (see Fig. 5.12), the members of the family are categorized into parents, spouses and children. Each entry is contained in separate cards, having their own set of fields. If the user has permission to edit, a button is shown, enabling the modification of the the priest's profile through the interface shown in Figure 5.10. In editing mode, fields can be modified, entities (cards) can be added or removed in any of the categories, and files or pictures can be uploaded or deleted.

Conclusion and further development

So far, the progression of the project proved successful. All of the main functionalities initially described by the client are part of the software platform. The convenient handling of the priest and parish records, and in particular their extensive searchability are the main achievements of the development activity. Moreover, the multi-tier account system with the approval-based registration system satisfies all the security and access-control related expectations. Regarding the non-functional requirements, the webpage has a clean user interface and intuitive usability.

The developers got a free hand over the exact realization of the client's ideas, therefore a few additional features were added as well, which were not explicit requirements. These only served the improvement of the developer duo's programming and technical skills. The map-based parish searching page and the well-received chart designer component are two such results of the creative freedom licensed to the author of the thesis.

Nevertheless, there are a more than a handful of tasks still on the kanban board. The list of the upcoming enhancements include the introduction of full-text searches in all profile data and the possibility to save a priest's profile page as a PDF.

The usage of the maps and data visualizations throughout the components can be increased as well. Parish profiles could feature a tab with preconfigured visualizations generated only from the local priests' data, while priest profiles are planned to include a map that compiles all the locations important in the life of the individual.

A few minor technical fixes are also planned, such as incorporating API tests into the *continuous integration* and optimizing error propagation.

Bibliography

[1] G. Acharya. *How to Send Email through Gmail SMTP Server Using GO*. Jan. 2, 2020. URL: https://pepipost.com/tutorials/send-email-through-gmail-smtp-server-using-go/ (visited on: 2020. May 3.)

[2] N. Babich. *Most Common UX Design Methods and Techniques*. July 13, 2017. URL: https://uxplanet.org/most-common-ux-design-methods-and-techniques-c9a9fdc25a1e (visited on: 2020. Apr. 28.)

[3] K. Ball. *The increasing nature of frontend complexity*. Jan. 30, 2019. URL: https://blog.logrocket.com/the-increasing-nature-of-frontend-complexity-b73c784c09ae/ (visited on: 2020. Apr. 30.)

[4] E. Boersma. *Docker Image vs Container: Everything You Need to Know*. May 3, 2019. URL: https://stackify.com/docker-image-vs-container-everything-you-need-to-know/ (visited on: 2020. June 5.)

[5] F. Copes. *What is a Single Page Application?* Nov. 11, 2018. URL: https://flaviocopes.com/single-page-application/ (visited on: 2020. Apr. 28.)

[6] A. Edwards. *An Overview of Go's Tooling*. Apr. 15, 2019. URL: https://www.alexedwards.net/blog/an-overview-of-go-tooling (visited on: 2020. Apr. 30.)

[7] S. Eskildsen. *Structured, pluggable logging for Go*. URL: https://github.com/sirupsen/logrus (visited on: 2020. Apr. 30.)

[8] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002, pages 601–606.

[9] S. Francia. *Go configuration with fangs*. URL: https://github.com/spf13/viper (visited on: 2020. Apr. 28.)

[10] A. Ghezala. *You don't have to use Redux*. June 1, 2019. URL: https://dev.to/anssamghezala/you-don-t-have-to-use-redux-32a6 (visited on: 2019. Jan. 30.)

[11] L. Gupta. *What is Golang and how to install it*. Nov. 17, 2019. URL: https://medium.com/datadriveninvestor/what-is-golang-and-how-to-install-it-e85002ab0871 (visited on: 2020. Apr. 28.)

[12] S. Lahoti. *Why Golang is the fastest growing language on GitHub*. Aug. 9, 2018. URL: https://hub.packtpub.com/why-golan-is-the-fastest-growing-language-on-github/ (visited on: 2020. Apr. 25.)

[13] P. Le Cam. *Introduction - React-Leaflet*. URL: https://react-leaflet.js.org/docs/en/intro (visited on: 2020. Apr. 30.)

[14] J. van Mourik. *GUI for editing your i18n translation files*. URL: https://github.com/jcbvm/i18n-editor (visited on: 2020. May 3.)

[15] D. Pásztor. *UX Design*. UXStudio, Budapest, 2016.

[16] J. Potter. *Compare NPM package downloads*. URL: https://www.npmtrends.com/@angular/core-vs-angular-vs-react-vs-vue-vs-ember-source (visited on: 2020. June 5.)

[17] T. Rascia. *Build a CRUD App in React with Hooks*. Nov. 7, 2018. URL: https://www.taniarascia.com/crud-app-in-react-with-hooks/ (visited on: 2020. Apr. 25.)

[18] M. Rehkopf. *What is a Kanban Board?* URL: https://www.atlassian.com/agile/kanban/boards (visited on: 2020. May 3.)

[19] I. Sacolick. *What is CI/CD?* Jan. 17, 2020. URL: https://www.infoworld.com/article/3271126/what-is-cicd-continuous-integration-and-continuous-delivery-explained.html (visited on: 2020. May 3.)

[20] J. Stackhouse. *Cross Origin Resource Sharing middleware for gin-gonic*. URL: https://github.com/itsjamie/gin-cors (visited on: 2020. Apr. 25.)

[21] W. Wattearachchi. *Testing with Jest and Enzyme in React*. A series with six parts. URL: https://blog.usejournal.com/testing-with-jest-and-enzyme-in-react-part-6-snapshot-testing-in-jest-72fb0ce91c5a (visited on: 2020. Apr. 30.)

[22] R. Wieruch. *How to fetch data with React Hooks?* Mar. 7, 2019. URL: https://www.robinwieruch.de/react-hooks-fetch-data (visited on: 2020. Apr. 25.)

[23] Axios official GitHub page. Promise based HTTP client for the browser and node.js. URL: https://github.com/axios/axios (visited on: 2020. Apr. 30.)

[24] Balsamiq official webpage. Balsamiq. Rapid, effective and fun wireframing software. URL: https://balsamiq.com/ (visited on: 2020. Apr. 30.)

[25] Docker reference documentation. Overview of Docker Compose. URL: https://docs.docker.com/compose/ (visited on: 2020. May 3.)

[26] Format.JS reference documentation. react-intl: Overview. URL: https://formatjs.io/docs/react-intl (visited on: 2020. Apr. 30.)

[27] Gin Web Framework reference documentation. Introduction. URL: https://gin-gonic.com/docs/introduction/ (visited on: 2020. Apr. 28.)

[28] GitKraken official webpage. GitKraken Git GUI. URL: https://www.gitkraken.com/git-client (visited on: 2020. May 3.)

[29] Golangci-lint official GitHub page. Linters Runner for Go. URL: https://github.com/golangci/golangci-lint (visited on: 2020. May 3.)

[30] Google. *App Maker*. URL: https://developers.google.com/appmaker/ (visited on: 2020. May 20.)

[31] Google Maps Platform reference documentation. Place Search. URL: https://developers.google.com/places/web-service/search (visited on: 2020. May 13.)

[32] KMDSZ. *23rd ETDK results*. URL: https://etdk.kmdsz.ro/general.php?id=21 (visited on: 2020. June 8.)

[33] Microsoft. *Power Apps*. URL: https://powerapps.microsoft.com/ (visited on: 2020. May 20.)

[34] MongoDB official GitHub page. The Go driver for MongoDB. URL: https://github.com/mongodb/mongo-go-driver (visited on: 2020. Apr. 28.)

[35] MongoDB reference documentation. Structure your Data for MongoDB. URL: https://docs.mongodb.com/guides/server/introduction/ (visited on: 2020. Apr. 25.)

[36] Oracle reference documentation. What Are RESTful Web Services? URL: https://docs.oracle.com/javaee/6/tutorial/doc/gijqy.html (visited on: 2020. Apr. 28.)

[37] Palantir Technologies. TSLint: An extensible linter for the TypeScript language. URL: https://github.com/palantir/tslint (visited on: 2020. May 3.)

[38] Postman official webpage. The Collaboration Platform for API Development. URL: https://www.postman.com/ (visited on: 2020. May 3.)

[39] React reference documentation. Introducing hooks. 2018. URL: https://reactjs.org/docs/hooks-intro.html (visited on: 2020. Apr. 25.)

[40] React reference documentation. Reconciliation. URL: https://reactjs.org/docs/reconciliation.html (visited on: 2020. Apr. 28.)

[41] React Training reference documentation. React Router: Declarative Routing for React.js. URL: https://reacttraining.com/react-router/web/guides/primary-components (visited on: 2020. Apr. 30.)

[42] Reaviz reference documentation. React Router: Declarative Routing for React.js. URL: https://reaviz.io/?path=/story/docs-intro--page (visited on: 2020. Apr. 30.)

[43] Scrum reference documentation. What is Scrum? URL: https://www.scrum.org/resources/what-is-scrum (visited on: 2020. May 3.)

[44] Semantic UI React reference documentation. The official Semantic-UI-React integration. URL: https://react.semantic-ui.com (visited on: 2020. Apr. 25.)

[45] Semantic UI React reference documentation. Theming. URL: https://react.semantic-ui.com/theming (visited on: 2020. Apr. 30.)

[46] Slack official webpage. *Where work happens*. URL: https://slack.com/ (visited on: 2020. June 10.)

[47] VS Code official webpage. Visual Studio Code - Code Editing. Redefined. URL: https://code.visualstudio.com/ (visited on: 2020. May 3.)